SpecCharts : A Language for System Level Specification and Synthesis

Frank Vahid
Sanjiv Narayan
Daniel D. Gajski

Technical Report #90-19
August 2, 1990

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-7063

narayan@ics.uci.edu
vahid@ics.uci.edu

Abstract

SpecCharts is a new language intended for system level specification and synthesis. SpecCharts represent a multi-module system with a hierarchy of state diagrams, catering to the expression of concurrent behavior, and using VHDL sequential statement semantics to describe a leaf state's (a state not composed of substates) functionality. The language permits protocol based data transfer, estimations, constraints, and state based description, all of which enable the omission of detail and thus enhance the comprehension of a system's behavior. The language is intended to represent a design throughout the system and chip levels of synthesis, i.e. converting an abstract specification into a set of one or more interconnected chips/modules, each having a well defined structure or being bound to a prefabricated component. Since good system design requires an executable specification language, SpecCharts can be simulated via automatic conversion to VHDL.

Contents

List of Figures

1 Introduction

System level synthesis refers to synthesis at the computer system level. The primary aim is to convert a system's specification into a set of one or more interconnected chips/modules. This involves determining the number of chips necessary to satisfy the given constraints (*estimation*), distributing the specifications among the chips (*partitioning*), finding a well-defined structure for each chip, and creating proper interchip/intermodule interfaces (*interface synthesis*).

SpecSyn is a synthesis tool being developed for such a purpose. The tool requires an executable specification language with appropriate abstractions to precisely and easily specify a design through system and chip levels of synthesis. The SpecCharts language had been created for this task. It permits description as a hierarchical state diagram, with leaf state functionality being described using VHDL sequential statements. Features include protocol based data transfer, specification of estimates and constraints, binding of chips/modules to prefabricated components, arbitrated signals, and several constructs that aid state based specification; these combined with the language's graphical/textual representation enable a system's behavior to be easily discerned and aid synthesis. A specification may be partial or complete, and on many different levels. For example, one could specify a portion of a system using a state, providing only estimates for that state such as the chip area or execution time. Or one could provide the exact behavior of that state by specifying substates, data structures, algorithms, dataflow statements, structural descriptions, or some combination thereof.

VHDL would seem to be a likely candidate language to use. It is a widely accepted standard and thus provides a common means for information exchange and the use of powerful tools (e.g. simulators, synthesizers, etc.). The language itself is excellent for creating structural and RTL descriptions, as well as simple algorithmic descriptions. However, VHDL can not be tuned for *all* applications, and thus can be difficult and tedious to use for system level specification. For example, many higher level systems, such as a CPU system, are inherently state based and contain many data transfers. But VHDL does not provide any abstractions to simplify the description of such systems, and requires one to implement state activation and arc transitions manually using control signals, and to be concerned with low level data transfer using ports and signal assignments. However, in system level design one should be able to concentrate more on functionality rather than spending time specifying the low level details. Trying to build these abstractions out of VHDL constructs is difficult and often impossible. VHDL excels at describing hardware; **SpecCharts is intended for systems** . The domain of each language is simply different. A system specification should not contain the amount of detail VHDL requires; instead, we can synthesize those details, convert to VHDL, and then pass the VHDL description to a lower level synthesis tool. Designers do not usually think in terms of programming languages when designing systems; instead, they draw flow charts, boxes, arrows, and think of protocols, variables, etc. SpecCharts is an attempt to capture these conceptualizations. It is worthwhile to exploit many advantages of VHDL (like wide acceptability, availablity of simulators, etc.) by developing a representation on top of it, not a replacement for it. We have also tried to use VHDL syntax and semantics whenever possible, so that VHDL designers will already be familiar with much of SpecCharts.

The next section discusses some of the current approaches to system level design specification. Section 3 presents some of the features of the SpecChart language, in particular

protocol based data transfer and resource arbitration. Finally we present some of the advantages that make a SpecChart representation of a design attractive.

2 Previous Work

The first step in a system-level design environment is a specification language that will permit exploration of design alternatives. In this section we will be exploring a few of the existing frameworks that allow a system-level design specification.

2.1 Statemate

Statemate, developed by i-Logix [Ha88, Ha90], integrates three views of the system being designed - the functional view, the structural view and the behavioral view. Each of these views is captured by the Activity charts, the Module charts and the State charts [Ha87]. Thus Statemate uses visual formalisms to describe the proposed system.

Activity charts provide the functional view of the system. It identifies the system's activities and processes, the data stores, and the information and control items that can potentially flow among them and between them and the environment. Statecharts describe the system behavior and identifies the conditions and events that would cause transitions between the various states of the system, as well as the actions to occur during transitions or when entering or exiting a state. Statecharts are capable of expressing hierarchy and concurrency. Module charts provide the structural perspective of the system by associating activities with physical components, and depict the communication channel between the components. Together, the three views form a specification model of the system which can be used in simulation to examine the performance of the system. Once the designer is satisfied with them, the three graphical views become a system specification, which may be used as a model against which to measure system implementation. A fourth language, the forms language, is needed to specify lengthy definitions, type and structure of data items, etc.

Major advantages of this approach include the use of state based constructs, which greatly simplify the specification, and the hierarchy and concurrency constructs which quite naturally decompose the system into modules. Disadvantages include the necessity of four languages to represent the system, since their correspondence with one another can be quite difficult to follow. Also, the method of specifying actions is insufficient, since often algorithms, which do not lend themselves easily to graphical constructs, need to be specified (thus requiring some sort of programming language).

2.2 ADAS - Architectural Design and Assesment System

ADAS [FrFI85, FrSC85], developed by the Research Triangle Institute, requires two types of digrams to specify a system - a dataflow diagram similar to the Statemate Activity chart and a block diagram to define the system hardware. In the ADAS environment, the designer first models the software without considering the hardware on which the functions will be implemented. Once the designer is satisfied that with the basic software functions, various hardware architectures may be defined. The designer may map software functions to various hardware resources and simulate the system to see how the system runs using different configurations.

A disadvantage to this approach is that a dataflow diagram is the starting specification. At the system level, a higher level specification is necessary.

2.3 Uninterpreted Modeling Using VHDL

In [HaAy89], a model is discussed which uses VHDL in a manner suitable for the system level. The model is uninterpreted, meaning that only information is passed (tokens), without meaning or form. Previous attempts, such as queuing models and petri nets, have failed across the full range of abstraction (behavioral, structural, etc.). The approach using VHDL does not fail in this regard since VHDL can represent at various levels of abstraction. The uninterpreted model uses two types of inputs, data and control, and two types of outputs, those that can overwrite existing tokens and those that must wait. A token is defined as a record, its fields being used for handshaking, indicating presence or absence of information, indicating identity, and other information. The model is then built, and simulation can then be done to determine certain characteristics of the model. Since VHDL is designed for interpreted models (presence of functions mapping inputs to outputs), a model can be uninterpreted, interpreted, or a hybrid, all using the same language. Succesive refinment can be done on a model to provide more interpretation while using the same VHDL environment, thus providing a single design path.

A major advantage of this approach is the use of the VHDL environment throughout; thus no new language needs to be learned, just the new modeling style. However, writing and understanding the model can be quite difficult, since the textual uninterpreted VHDL model does not make the functionality easily discernible. Also, state based constructs are not present. Perhaps future improvements to this modeling technique will address these issues.

2.4 CASE Languages

Many other languages have been developed for software system specification, including VDM ([Jo86]), Z ([Ha86]), Larch ([GuHoWi85]), OBJ ([GoTa79]), ESML ([BrJe88]), RSL ([Me88]), PAISLey ([Me88]), and SCR ([Me88]). Each is based on some combination of algebraic expressions, mathematical entities such as sets or functions, propositional calculus, predicate calculus, procedural languages, Petri nets, data flow diagrams, or transformation schema. Most are well suited for specifying the requirements of pure software, but are difficult and ackward to use for describing mixed hardware/software computer systems, and can not describe hardware systems at all, which is necessary if a single language is to be used throughout system synthesis. Many suffer from a lack of hierarchy or concurrency, while others are quite mathematical and thus prevent easy understanding of new specifications resulting from synthesis steps. None permit the powerful abstraction of hierarchical/concurrent state diagrams.

SpecCharts is an attempt to solve the above problems. Firstly, the SpecChart language combines the three views of a specification (functional, behavioral and structural) in a single graphical format. While this could have the disadvantage of cluttering the specification, it prevents having to mentally resolve separate specifications into one. We have found that cluttering is not a problem for the computer systems we have modeled.

Secondly, the SpecCharts language was designed with the goal of *system-level synthesis* in mind. Therefore it provides a variety of constructs which would aid the various synthesis

tasks. While resembling StateCharts in its representation of hierarchy and concurrency, it also contains numerous constructs which are paricularly useful when dealing with system level designs. Both of these types of constructs will be introduced in the following sections.

Perhaps the most significant difference between most of the specification methods described above and SpecCharts is that SpecCharts is built on top of a widely accepted hardware description language - VHDL, thus simplifying its use by hardware designers. Algorithms or simple functionality are represented using VHDL and incorporated directly into the specification.

SpecCharts are thus a combination of (1) the system modeling power of StateCharts' hierarchy/concurrency extensions to state diagrams, (2) the hardware modeling power and familiarity of VHDL, and (3) new constructs intended to simplify computer system description and to aid system level synthesis.

3 The SpecCharts Language

The SpecCharts language is introduced through an example. In Figure 1, a block diagram of an example computer system is shown. In Figure 2, a *partial* SpecChart of the system is shown. At the topmost level, the specification consists of the state SYSTEM, which consists of three concurrent states - CLOCK_CHIP, KEYBOARD_INTERFACE and PROCESSOR. PROCESSOR in turn consists of a CPU similar to the Intel 8086, a DMA controller such as the Intel 8237, and a memory.

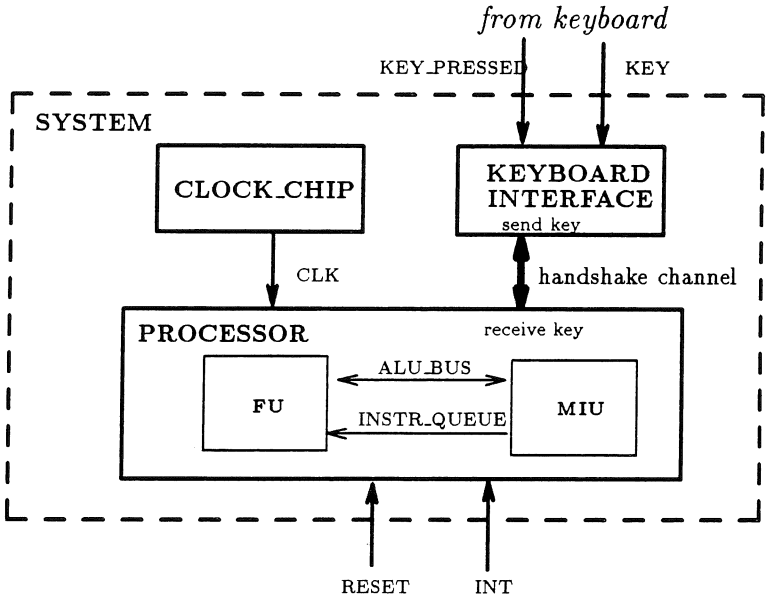


Figure 1: Block diagram of an example computer system

From the example, it can be seen that the description consists of hierarchical state diagrams with sequential or concurrent substates. Graphical conventions that apply to any state at any level of hierarchy include:

- a state is represented as a box

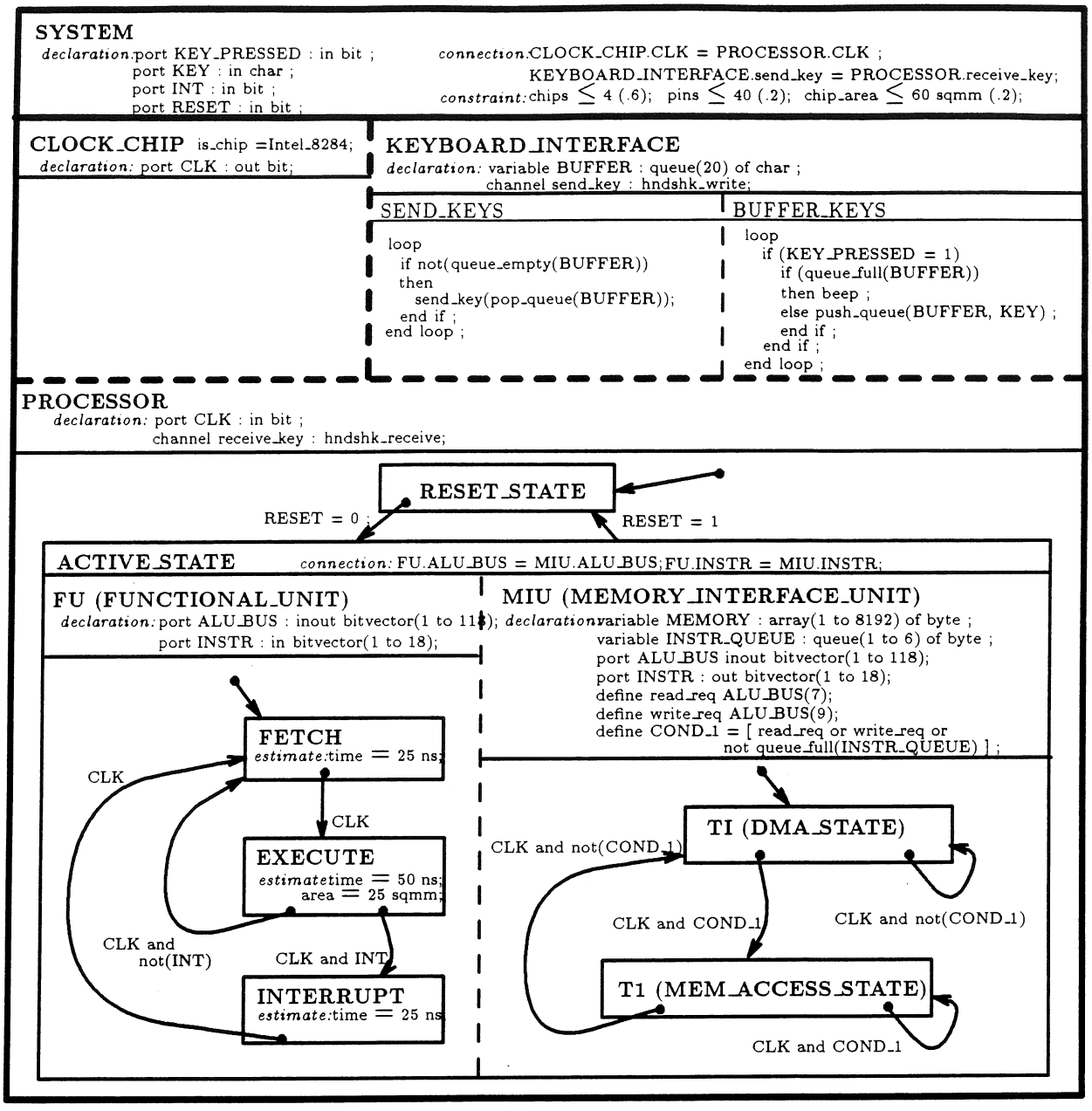


Figure 2: A Partial SpecChart of a Computer System

- sequential substates are represented as states with transition arcs between them
- concurrent substates are represented as states separted by dotted lines

Leaf states contain code, i.e. VHDL sequential statements. Since the SpecCharts description consists of a hierarchy of states, a description of the language used to represent a single state is sufficient to describe the language itself.

As in state diagrams, we can either be 'in' or 'not in' a state. Specifically, a state may be:

- **Active**– we are 'in' the state, and are either:
  - *executing*– The state is performing computations. If a leaf state, the VHDL statements are being executed. If a non-leaf, one or more of the state's substates is active.
  - *complete*– The state has finished its computations but has not yet been deactivated. If a leaf state, the end of the VHDL statements has been reached AND all signals assigned in this state have received their new values. If a non-leaf, all substates are inactive and control has flowed to a special 'stop' dot from where no substate can be reactivated.
- **Inactive**– we are not in the state

Each state consists of sections, each of which will now be discussed.

3.1 Program Section

This specifies the actions carried out by a state, i.e. its functionality. It can be described in one of three ways: sequential substates, concurrent substates, or code. This is shown in Figure 3.

3.1.1 Sequential Substates

Only one sequential substate is active at a time. When a state containing sequential substates is activated, the initial substate is activated. The initial substate is pointed to graphically by an arc (without a condition) coming from the special 'start' dot. There are two types of arcs that may be used to sequence between sequential substates:

- Exit-on-completion (**EOC**) arc : traverse the arc if the source substate has *completed* and the associated arc condition is true, where the condition may be any boolean expression. The arc is graphically shown as originating from a bold dot inside the substate.
- Exit-immediately (**EI**) arc : traverse the arc if the source substate is *active* and the associated arc condition is true, where the condition may be any expression that can occur in a VHDL 'wait until' clause. The arc is graphically shown originating from a substate's perimeter.

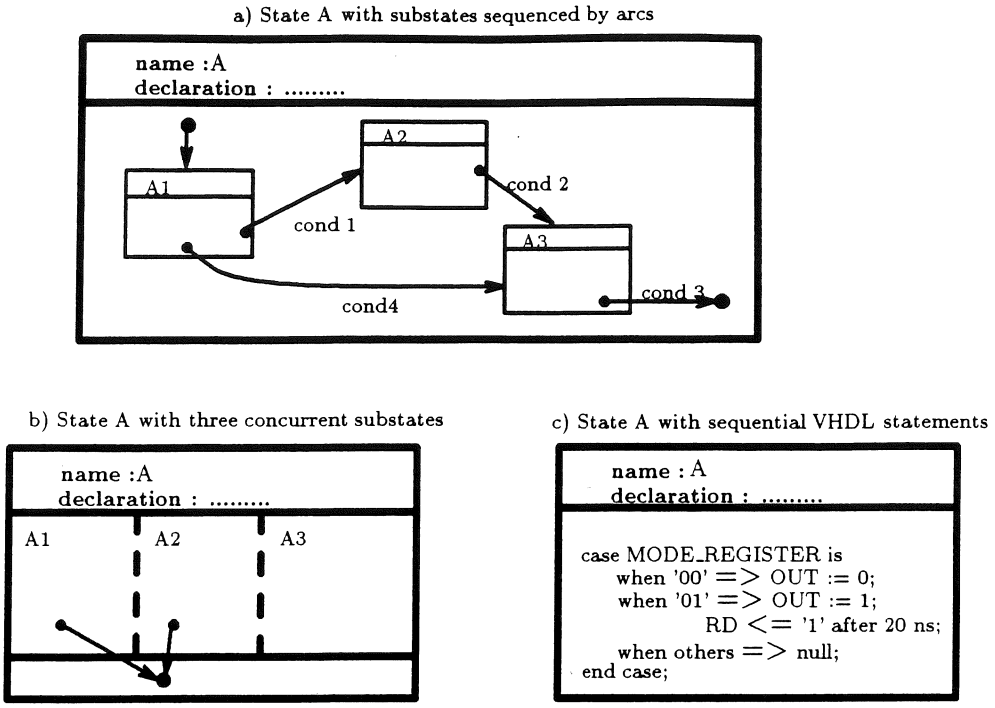


Figure 3: Types of State Organizations supported by SpecCharts

Whenever an arc is traversed, the source substate is deactivated and the destination substate is activated. A substate can have any number of arcs exiting it or pointing to it. If more than one arc could be traversed at a given time, priority is counterclockwise; alternatively the arcs may be numbered. If more than one EOC arc exits a substate, one of them may have the condition 'other' associated with it, which is true only if all other EOC arc conditions of this substate are false. An EOC arc with no condition associated with it by default has the condition 'true'. An EI arc may have the expression 'timeout(x)', where x is a time, associated with it, meaning that x amount of time after entering the substate, the expression becomes true so the arc should be traversed. The state completes when an arc pointing to the 'stop' dot is traversed (this is optional, i.e. the state does not have to complete).

Figure 4 shows how a clock signal of period 100 ns can be gen erated using different types of arcs. In Figure 4a, the transitions occur after the *wait for 50 ns* statement has been executed. No condition is associated with the arc, and thus assumed to be true at all times. In Figure 4b, the last statement assigning a value to the CLK signal also triggers the exit_immediately arc transition. Figure 4c shows how timeout arcs can be used to achieve the same result.

In Figure 2, state FU has three sequential substates: FETCH, EXECUTE, and INTERRUPT. EOC arcs are used to sequence between them; thus, when the FETCH state has completed AND the clock is rising, EXECUTE is activated. An example of EI arc is the arc labeled "RESET = 1", which results in a transition to the RESET state from the ACTIVE

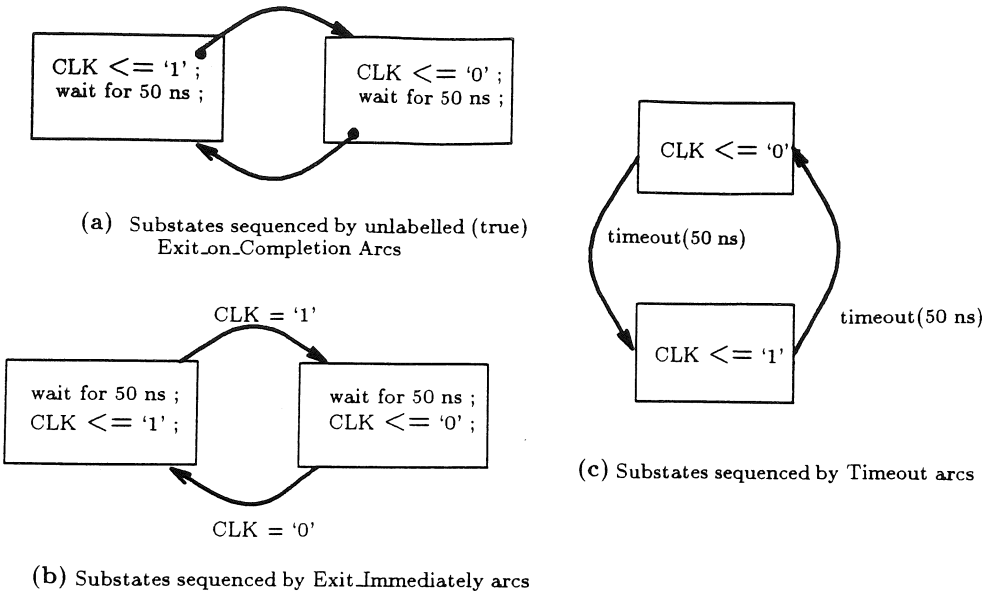


Figure 4: Generation of a clock signal using different types of arcs

state regardless of the execution status of the ACTIVE state.

3.1.2 Concurrent Substates

When a state containing concurrent substates is activated, *all its substates are activated concurrently.* For example, in Figure 2, the state SYSTEM consists of three concurrently executing substates - CLOCK_CHIP, KEYBOARD_INTERFACE, and PROCESSOR.

A state with concurrent substates may complete, but how so is more subtle than for sequential substates. One or more of the substates may contain an EOC arc with the condition 'true' pointing to a 'stop' dot; when all of these arcs have been 'traversed' (i.e. when all those substates have completed), the state is complete.

For both sequential and concurrent program organizations, if the parent state is deactivated (by an EI arc), all of its active substates are also deactivated.

3.1.3 Code (Leaf State)

Instead of using another state diagram, a state's functionality can be described using code, i.e. VHDL sequential statements (those that can occur in a VHDL process body). These statements include the *wait, assertion, signal/variable assignment, if, case, loop, next, exit, return* and *null*statements and *procedure calls* . When the state is activated, execution starts at the beginning of the code. If execution reaches the end of the code, or more specifically runs off the end of the code, AND all signal assignments made in the code have taken place, then the state is complete. For example, a state containing only the code *M(PC) <= AC after 10 ns* completes 10 ns after it was activated. Code may not necessarily complete, for example if it is enclosed in a loop.

If a leaf state is deactivated (by an EI arc), not only is execution stopped, but all signal

transactions scheduled by this state to occur in the future are removed, so that no further updates are made. For example, if a state containing *M(PC) <= AC after 10 ns* is deactivated 5 ns after having been activated, M(PC) does *not* get the value of AC 5 ns later.

3.2 Declaration Section

Each state may contain declarations, whose scope is all descendant states (all substates at any depth of hierarchy) as well as the state's program section. Declarations are the same as VHDL declarations, with the following exceptions:

- Channels can be declared as follows: *channel channelname : protocolname;*
  They are high level abstractions used to avoid having to specify low level ports and signal assignment statements for data transfers between concurrent states. See section 3.7.
- macros can be declared similar to those in C, as follows: *define macroname anytext;*
- No access types, file types, file objects, interface declarations, or component declarations are needed.
- Ports are declared as follows:
  port identifier_list: mode subtype_indication :=static_expression. Mode must be in, out, or inout.
- Signal declarations have been exanded to permit abitrated signals See section 3.8.

Global variables can not be accessed in concurrent substates, for the same reason that global variables are not permitted in VHDL (i.e. a global variable's value is not defined over time) . However, variables may be accessed by sequential substates, since only one can be active. Variables can be *declared* as global over concurrent substates, just not *accessed* in more than one of them.

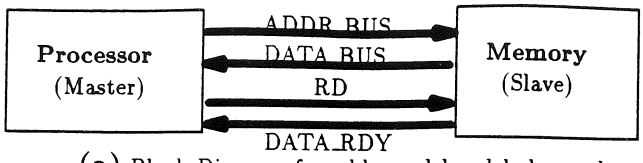
3.3 Connection Section

This section specifies connections between ports/channels belonging to concurrent substates, between ports/channels of a state with ports/channels of one of its concurrent substates, or between the ports/channels of a state with other ports/channels of the same state. A connection is specified as *port_1 : port_2 : ... : port_n;* Figure 2 has a connection specified for the ALU_BUS ports on the FU and the MIU.

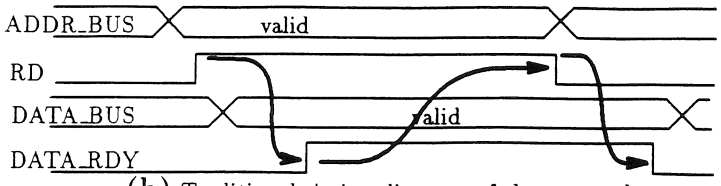
3.4 Name Section

This section contains the name of the state, certain optional predefined attributes, and any parameters (for state macros). A state macro is a state that is stored in a design library, and can be used as a substate in another SpecChart. The optional parameters would be replaced by name. For example, a simple state implementing a clock generator might have parameters indicating the clock's high time and low time, which could thus be used in different SpecCharts without needing to rewrite the state.

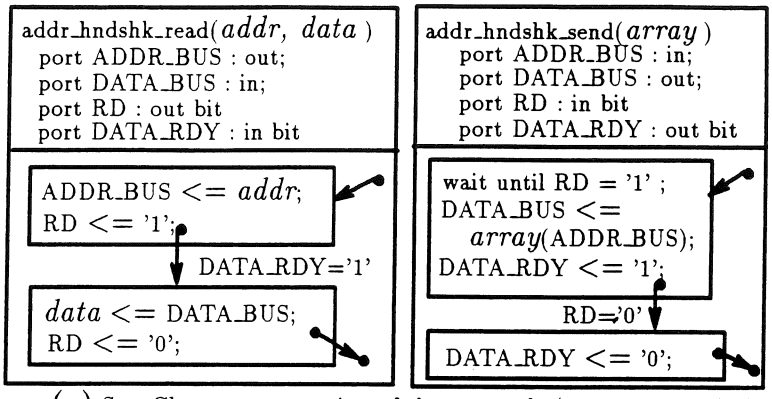
Attributes provide additional information about the state. The *is_chip* attribute indicates that the state makes up a single chip. This can be done by the user to guide a synthesis tool,



(a) Block Diagram for addressed handshake read



(b) Traditional timing diagram of the protocol



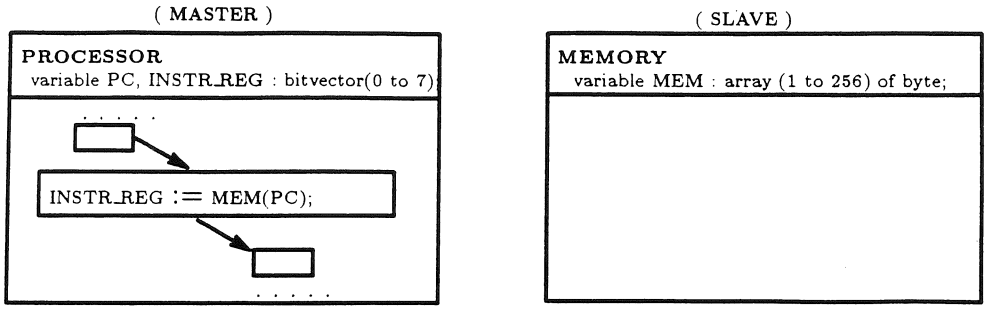
(c) SpecChart representation of the protocols (parameters in italics)

Figure 5: Representation of the Address Handshake Protocol

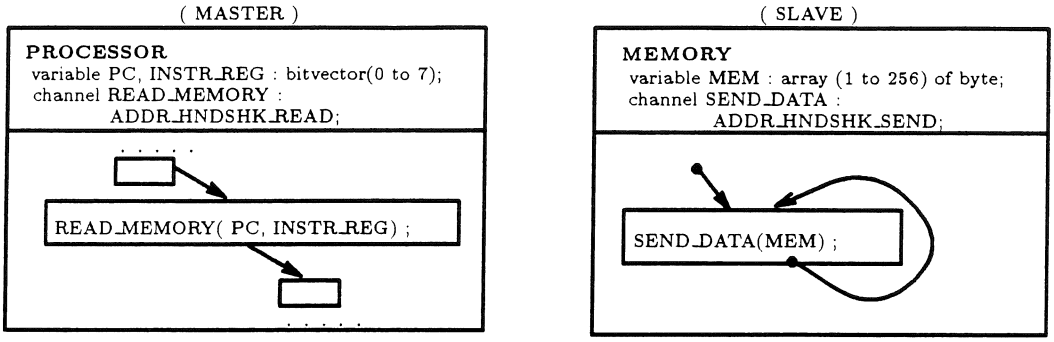
signals, similar to resolved signals. When an arbitrated signal is assigned, the arbitration scheme determines who gets access. This greatly reduces the amount of code needed, and keeps the intended functionality highly visible. A resolved but unarbitrated signal can be thought of as having an arbiter that always lets all processes have access rights. The details of the arbitration scheme can be synthesized later.

3.9 Making SpecCharts Simulatable

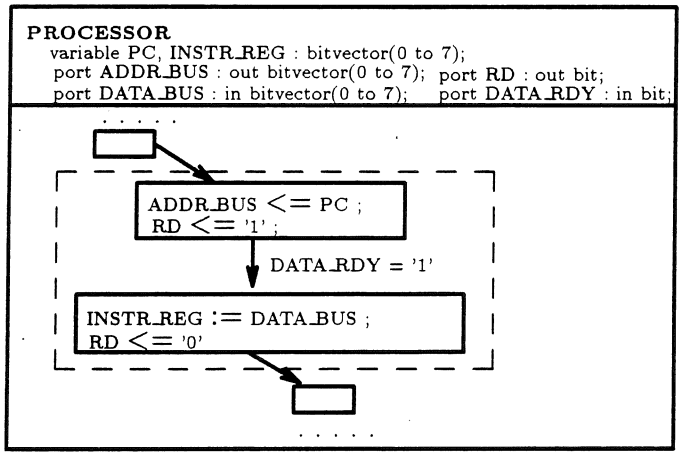
An essential requirement of system design is that the specification language be executable, so that functional verification as well as system performance can be analyzed. Our approach is to convert SpecCharts to VHDL [NaVa90b], and then use one of the many VHDL simulators available. Since SpecCharts use VHDL syntax and semantics whenever possible, the translation is greatly simplified. The major task is to convert the abstractions present in SpecCharts, such as state based description constructs and protocol based data transfer, to functionally equivalent VHDL. Converting to VHDL, rather than implementing a new simulator, greatly reduced the time needed to make SpecCharts simulatable. The general translation scheme uses nested blocks, with processes at the leaf level, and control processes for each state to properly activate and deactivate substates. The output of the translator is a VHDL entity, which can be used as any other VHDL entity.



(a) Diagram showing inter-process access of data-structure MEM



(b)Diagram showing channel declaration and parameterized protocol instantiation



(c) Diagram showing protocol expansion with actual parameters and ports after interface synthesis

Figure 6: Protocol instantiation and expansion

4 Suitability of SpecCharts for System Level Description and Synthesis

In the preceding sections we have introduced the SpecCharts language. We now evaluate the suitability of SpecCharts as a language for system level specification and synthesis, while contrasting it with VHDL.

4.1 State based System Specification

Designers often describe systems using some combination of flow charts, state diagrams, and pseudocode. SpecCharts are intended to capture these conceptualizations, by a combination of hierarchical state diagrams and VHDL code. They are well suited for decomposing the functionality of a system into a collection of nested sequential and concurrent substates, and thus can concisely specify real systems in an easy to manage and understand fashion. The concepts of EOC and EI arcs have also proven to simplify the description of such systems.

The concepts of states, state hierarchy, arcs, etc., are not present in VHDL, and thus sequencing details must be explicitly specified. This can prove to be cumbersome, can introduce errors, and reduce the models understandability when modeling systems.

4.2 Protocol Based Data Transfer

Data transfer can occur very frequently in a design and a single access of a data structure may consist of several operations (e.g. a handshake) which have to be performed for every access. In SpecCharts, channels are used to perform complex data transfers. The functionality is kept understandable since the channel's associated protocol describes the data transfer in high level terms. The SpecChart definition of the protocols not only permits specification of complex protocols, but simplifies synthesis of the details of the data transfer.

4.3 Arbitrated Signals

While the concept of resolved signals is necessary at the hardware level, arbitrated signals are more suited to the system level. They very concisely specify behavior that would take many additional statements and processes to represent explicitly, thus keeping the intended functionality easily discernible. The definition of arbitrated signals permits easy conversion to an equivalent, but much more detailed, SpecChart during synthesis.

4.4 Partial Design Specification

It is essential that a system level specification language permit partial design, i.e. omission of the internal details of some (or all) states, so that higher level synthesis decisions (e.g. partitioning) can be performed. To provide for this very likely scenario, SpecCharts has an Estimates Section. This enables the designer to delay a detailed specification of a state in the design by providing estimates of the design parameters, which are then used by the SpecSyn estimation tools directly.

4.5 Varying Degree of control over synthesis

Constraints may be specified, along with relative importances, to guide synthesis tools. Though constraints are confined to those of a state, and thus arbitrary constraints are not permitted [AmBoWi89], we believe this is sufficient for system level design. The designer can specify a group of states to be synthesized into one chip using the *is_chip* attribute, or can bind a state to a pre-fabricated chip/component, and consequently no synthesis is carried out on that part of the design. User provided estimations can also influence synthesis steps.

5 Results and Future Work

The current graphical interface is an X widget based application (a complete graphical interface has not been implemented). SpecCharts can be entered with this system and stored in files in a textual format, or directly entered in their textual format using any text editor. The system implementing the SpecCharts language is written in C and runs on Sun3/Sun4 workstations under UNIX. It consists of a library of C routines for parsing the textual format, maintaining and manipulating the SpecChart internal representation, and performing basic error checking, and provides a somewhat object oriented environment. This library is intended for use by various SpecChart applications, such as the SpecChart to VHDL translator [NaVa90b] or partitioner. The SpecChart to VHDL translator has also been implemented. It takes as input the SpecChart textual files and outputs a file containing a simulatable VHDL entity. The translator is very fast, translating large SpecCharts into VHDL in the order of a few seconds. The ratio of the amount of SpecChart textual code to generated VHDL code is approximately 1:2.5.

Using this platform, we have modeled and simulated several examples. One detailed example is Armstrong's Controlled Counter [Arms89, NaVa90a]. Entering the SpecChart via the graphical interface required 67 lines of code. The textual SpecChart created by the graphical application contained 139 lines, and was translated to 271 lines of VHDL in .3 seconds. A second detailed example is the DRACO peripheral interface ASIC [Rock89, NaVa90a]. This example is particularly interesting as it is a real existing chip used in industry. The specifications were provided by Rockwell International, and a detailed SpecChart model was created from this specification and from information provided by several Rockwell engineers. To ensure that the SpecChart model was correct and complete, we used Rockwell's original test vectors, consisting of 23,000 lines of process code (the process statements were converted from VTI format statements to VHDL using an automatic converter). The simulation verified that the SpecChart model of DRACO behaved correctly. The SpecChart model required 209 lines of code. This can be compared to a handwritten VHDL model of DRACO which required 392 lines of VHDL code to provide the same functionality. The 245 line textual SpecChart model was translated to 510 lines of VHDL in 1.2 seconds by the SpecCharts to VHDL translator. Both the SpecChart and handwritten VHDL models were tested with the same test set and the yielded identical results.

To further test the modeling ability of SpecCharts, we will be modeling the Intel8086 processor chip. We will then refine our synthesis algorithms, and begin implementing real synthesis (e.g. partioners, estimators, interface synthesizers) on top of the current SpecCharts platform.

6 Conclusion

Specification at the system level requires that less detail need exist in the description. It is the job of synthesis to provide these details. SpecSyn provides the SpecCharts specification language to represent the design at the system level. The language is executable and contains several abstractions and constructs that simplify specification at this level, such as state based description, protocol based data transfer, and arbitrated signals, thus permitting a designer to concentrate on functionality as well as enabling easier understanding of the updates performed during synthesis steps. Many of the constructs are designed to permit easy synthesis of the lower level details. The results of several examples illustrated the ability of the language to concisely model real designs, and demonstrated the feasibility and advantages of building a representation on top of VHDL.

7 Acknowledgements

This work was supported by the Semiconductor Research Corporation (grant #90-DJ-146) and the National Science Foundation (grant #MIP8922851). We are grateful for their support. We would also like to thank Joe Lis and Tedd Hadley for their helpful suggestions.

8 References

[AmBoWi89] Amon, T., Boriello, G., Winder, W., "A Unified Behavioral/Structural Representation for Simulation and Synthesis", 1989

[Arms89] Armstrong, J., "Chip Level Modeling Using VHDL", Prentice-Hall, 1989.

[BrJe88] Bruyn, W., Jensen, R., Keskar, D., Ward, P., "ESML: An Extended Systems Modeling Language Based on the Data Flow Diagram", ACM Software Engineering Notes vol 13 no 1, January 1988.

[DuHaGa89] Dutt, N., Hadley, T., and Gajski, D., "BIF: A Behavioral Intermediate Format for High Level Synthesis", University of California, Irvine, Technical Report 89-03, September 1989.

[FrFI85] Frank, G., Franke, D., and Ingogly, W., "An Architectur e Design and Assessment System", VLSI Design, August 1985.

[FrSC85] Frank, G., Smith, C., and Cuadrado, J., "An Architectur e Design and Assessment System for SoftwareHardware Codesign", Proceedings of the 22nd Design Automation Conference, Las Vegas, Nevada, June 1985.

[GoTa79] Gougen, J., Tardo, J., "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications", Specifications of Reliable Software, 1979.

[GuHoWi85] Guttag, J., Horning, J., Wing, J., "The Larch Family of Specification Languages", IEEE Software 2 (5), Septemeber 1985.

[HaAy89] F.T. Hady, J.H. Aylor, et al, "Uninterpreted Modeling using the VHSIC Hardware Description Language (VHDL)", ICCAD, November 1989.

[Ha87] Harel, D., "Statecharts : A Visual Formalism for Complex Systems", Science of Computer Programming 8, 1987 pp 231-274.

[Ha88] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., and Shtul-Trauring, A., "STATEMATE : A Working Environment for the Developement of Complex Reactive Systems", Proceedings of the Intl. Conf. on Software Engineering, April 1988.

[Ha90] Harding, B., "Mixed CASE and CAE/CAD tools ease designers' headaches", Computer Design, January 1990.

[Ha86] Hayes, I., Specification Case Studies, Prentice-Hall, London, 1986.

[IEEE88] IEEE Standard VHDL Lanuage Reference Manual, IEEE, March 1988.

[Jo86] Jones, C., Systematic Software Development Using VDM, Prentice-Hall, London, 1986.

[LiSU89] Lipsett, R., Schaefer, C.F., and Ussery, C. "VHDL : Hardware Description and Design " Kluwer Academic Publishers, 1989.

[LiGa88] Lis, J., and Gajski, D., "Synthesis from VHDL", ICCD, 1988.

[MaWa90] MacDonald, R., and Waxman, R., "Operational Specification of the SINCGARS Radio in VHDL", AFCEA-IEEE Tactical Communications Conference, April 1990.

[Me88] Melhart, B., "Specification Languages for Embedded Systems: a Survey", University of California, Irvine, Technical Report 88-17, June 1988.

[NaVa90a] Narayan, S. and Vahid, F., "Modeling with SpecCharts", University of California, Irvine, Technical Report 90-20, June 1990.

[NaVa90b] Narayan, S. and Vahid, F., "Translating SpecCharts to VHDL", University of California, Irvine, Technical Report 90-21, July 1990.

[Rock89] Rockwell International, "DRACO Engineering Report", April 1989.

[VaNaGa90] Vahid, F., Narayan, S., and Gajski, D., "Synthesis from Specifications: Basic Concepts", University of California, Irvine, Technical Report 90-03, January 1990.

[Zyca89] Zycad Corporation, Menlo Park, CA 1989

A Current SpecChart Syntax

This appendix gives the current syntax which the SpecChart parser can parse and convert to an internal representation. While the syntax below is not the complete SpecCharts syntax, it is quite a large subset, perhaps covering 90% of the syntax. It is continually being updated to cover more.

```
state       : TSTATE TLCURL sections TRCURL
    ;

abstract_literal
            : unsigned_integer
            | TFLOAT
            | TBASEDLIT
    ;

actual_parameter
            : expression
            ;

actual_parameters
    : actual_parameter
            | actual_parameters TCOMMA actual_parameter
            ;

actual_parameter_list
            : actual_parameters
            ;

adding_operator
            : TPLUS
            | TMINUS
            | TCONCAT
            ;

                    /* arc_type */      /* cond */    /* next substate */
arc     : TLPAREN identifier TCOMMA expression TCOMMA identifier TRPAREN
    | TLPAREN identifier TCOMMA TTIMEOUT TLPAREN expression TRPAREN
              TCOMMA identifier TRPAREN
    | TLPAREN identifier TCOMMA TOTHER TCOMMA identifier TRPAREN
            ;

arcs        : arc
    | arcs TCOMMA arc
            | empty
            ;

assert_statement
            : TASSERT expression report_opt severity_opt TSEMICOLON
            ;

attrib_exp_opt
             /* NOTE: have to use brackets instead of parenthesis since
 the latter causes shift/reduce conflict in yacc
```

```
                (can't tell if it's  a'(attr(x)) or (a'attr)(x)) */
    : TLBRACK unsigned_integer time_unit TRBRACK
    | empty
    ;

attribute
            : TSTABLE attrib_exp_opt
            | TEVENT attrib_exp_opt
            | TACTIVE attrib_exp_opt
            | TQUIET attrib_exp_opt
            | TDELAYED attrib_exp_opt
            | TLOW
            | THIGH
            | TLENGTH
    ;

attribute_name
            : prefix TAPOSTROPHE attribute
    ;

binding     : TEQUAL identifier
            ;

case_statement
            : TCASE expression TIS
            ;

choices     : literal
            | choices TBAR literal
            ;

code
            : TCODE TLCURL statements TRCURL
            | TCODE TLCURL empty TRCURL
            ;

comment_statement
    : TCOMMENT
            ;

conc_ss     : TCONCURRENT_SUBSTATES TLCURL substates TRCURL
            | TCONCURRENT_SUBSTATES TLCURL empty TRCURL
            ;

condition_clause
            : empty
    | TUNTIL expression
            ;

connectee   : identifier TPERIOD identifier
            ;

connectees  : connectee
            | connectees TCOLON connectee
            ;

connection  : connectees TSEMICOLON
```

```
            ;

 connections : connection
             | connections connection
             ;

 connections_sct
             : TCONNECTIONS TLCURL connections TRCURL
             | TCONNECTIONS TLCURL empty       TRCURL
             | empty
             ;


 constraint  : trait TLTE unsigned_integer units TLPAREN
       unsigned_integer TRPAREN TSEMICOLON
             ;

 constraints : constraint
             | constraints constraint
             ;

 constraints_sct
             : TCONSTRAINTS TLCURL constraints TRCURL
             | TCONSTRAINTS TLCURL empty TRCURL
             | empty
             ;
declaration :
              TCHANNEL identifier TCOLON identifier TSEMICOLON
             | TCOMMENT
             | TCONSTANT  identifier
             | TDEFINE identifier
             | TFUNCTION function_name TLPAREN formal_parameter_list TRPAREN
               TRETURN type_mark TIS optional_declarations TBEGIN statements
               TEND  TSEMICOLON
             | TPORT identifier TCOLON mode subtype_indication signal_kind
               variable_init_asgn TSEMICOLON
             | TPROCEDURE procedure_name TLPAREN formal_parameter_list
               TRPAREN TIS optional_declarations
               TBEGIN statements TEND  TSEMICOLON
             | TSIGNAL identifier TCOLON subtype_indication signal_kind
               variable_init_asgn TSEMICOLON
             | TSUBTYPE identifier TIS subtype_indication TSEMICOLON
             | TTYPE identifier TIS type_definition TSEMICOLON
             | TVARIABLE identifier TCOLON subtype_indication
               variable_init_asgn TSEMICOLON
             ;

declarations
             : declaration
             | declarations declaration
             ;

optional_declarations
             : declarations
    | empty
             ;
```

```
declarations_sct
            : TDECLARATIONS TLCURL optional_declarations TRCURL
            | empty
            ;

direction   : TTO
            | TDOWNTO
            ;

discrete_range
            : subtype_indication
            /* currently restricted */
            | unsigned_integer direction unsigned_integer
            ;

discrete_ranges
    : discrete_range
            | discrete_ranges TCOMMA discrete_range
            ;

discrete_range_list
            : discrete_ranges
            ;

else_statement
            : TELSE
            ;
elsif_statement
            : TELSIF expression TTHEN
            ;

empty       : ;

endcase_statement
            : TEND TCASE TSEMICOLON
            ;

endif_statement
            : TEND TIF TSEMICOLON
            ;

endloop_statement
            : TEND TLOOP optional_end_label TSEMICOLON
            ;

enumeration_literal
            : identifier
            | TCHARLIT
            ;

enumeration_literals
            : enumeration_literal
            | enumeration_literals TCOMMA enumeration_literal
            ;

enumeration_literal_list
    : enumeration_literals
```

```
                ;

    exit_statement
                : TEXIT optional_end_label TSEMICOLON
                | TEXIT optional_end_label TWHEN expression TSEMICOLON
                ;


    estimate    : trait TEQUAL unsigned_integer units TSEMICOLON
                ;

    estimates   : estimate
                | estimates estimate
                ;

    estimates_sct
                : TESTIMATES TLCURL estimates TRCURL
                | TESTIMATES TLCURL empty TRCURL
                | empty
                ;

    expression  : relation
                | relation logical_operator expression
                ;

    factor      : primary
                | primary TPOWER primary
                | TABS primary
                | TNOT primary
                ;

    forloop_statement
                : optional_label TFOR identifier TIN unsigned_integer
                  direction unsigned_integer TLOOP
                ;


    formal_parameter_optional_bus
                : TBUS
                | empty
                ;

    formal_parameter_optional_class
                : TCONSTANT
                | TSIGNAL
                | TVARIABLE
                | empty
                ;


    formal_parameter_optional_mode
                : TIN
                | TINOUT
                | TOUT
                | empty
                ;
```

```
formal_parameter
            : formal_parameter_optional_class identifier TCOLON
              formal_parameter_optional_mode type_mark
              formal_parameter_optional_bus
            ;

formal_parameters  : formal_parameter
            | formal_parameters TSEMICOLON formal_parameter
            ;

formal_parameter_list
            : formal_parameters
            ;

function_call
            : function_name TLPAREN actual_parameter_list TRPAREN
            ;

function_name
            : TFCTNAME
            ;


identifier  : TIDENTIFIER
            ;

if_statement: TIF expression TTHEN
            ;

index_subtype_definition
            : type_mark TRANGE TBOX
            ;

index_subtype_definitions
            : index_subtype_definition
            | index_subtype_definitions TCOMMA index_subtype_definition
            ;

index_subtype_definition_list
    : index_subtype_definitions

indexed_name: prefix TLPAREN indexed_name_index TRPAREN
    ;

indexed_name_index
            : expression
            ;

is_chip     : TIS_CHIP
            ;

is_module   : TIS_MODULE
            ;

literal     : numeric_literal
            | TCHARLIT
            | TSTRINGLIT
```

```
            | TBSTRINGLIT
            ;


 logical_operator
            : TAND
            | TOR
            | TNAND
            | TNOR
            | TXOR
            ;

 loop_statement
            : optional_label TLOOP
            ;

 mode     : TIN
            | TOUT
            | TINOUT
            ;

 multiplying_operator
            : TTIMES
            | TDIVIDE
            | TMOD
            | TREM
            ;

 name       : simple_name
            | slice_name
            | indexed_name
            | attribute_name
            ;

 name_parameters
            : TLPAREN parameter_list TRPAREN
            | empty
            ;

 name_sct   : TNAME TLCURL identifier name_parameters stateattributes_opt TRCURL
            ;

 null_statement
            : TNULL TSEMICOLON
            ;

 numeric_literal
            : abstract_literal
    | physical_literal
    ;

 optional_end_label
            : empty
            | identifier
            ;

 optional_label
```

```
            : empty
            | identifier TCOLON
            ;

parameter   : identifier
            ;


parameters  : parameter
            | parameters TCOMMA parameter
            ;

parameter_list
            : parameters
            ;

physical_literal
            : abstract_literal time_unit
            ;

prefix      : name
            ;

primary     : name
            | literal
            | TLPAREN expression TRPAREN
            | function_call
            ;

procedure_call_statement
            : procedure_name TLPAREN actual_parameter_list TRPAREN TSEMICOLON
            ;

procedure_name
            : TPROCNAME      /* see WARNING at top of file */
              /* NOTE: this should be fixed.  Procedure names should be
                 any name, but we get a shift/reduce conflict from yacc.
                 Thus we currently restrict to identifier ending in _proc */
            ;


program_sct : code
            | seq_ss
            | conc_ss
            | empty
            ;

relation    : simple_expression
            | simple_expression relational_operator simple_expression
            ;

relational_operator
            : TEQUAL
            | TNEQUAL
            | TLT
            | TLTE
            | TGT
```

```
            | TGTE
            ;

 report_opt  : empty
             | TREPORT string
             ;

 resfun_opt  : empty
             | identifier
             ;

 return_statement
             : TRETURN TSEMICOLON
             | TRETURN expression TSEMICOLON
             ;

 sections    : name_sct  declarations_sct  connections_sct
               estimates_sct  constraints_sct  program_sct
             ;

 sensitivity_clause
             : empty
             | TON signal_name_list
             ;

 seq_ss      : TSEQUENTIAL_SUBSTATES TLCURL substates TRCURL
             | TSEQUENTIAL_SUBSTATES TLCURL empty TRCURL
             ;

 severity_opt: empty
             | TSEVERITY identifier
             ;

 sign        : TPLUS
             | TMINUS
             ;

 signal_assignment_statement
           /* currently restricted to only 1 target signal */
           /* currently restricted to only 1 waveform element */
            : name TLTE waveform TSEMICOLON
            | name TLTE TTRANSPORT waveform TSEMICOLON
            ;

 signal_kind :
               empty
             | TBUS
             | TREGISTER
             ;

 signal_name_list
             : name
             | signal_name_list TCOMMA name
             ;

 simple_expression
             : term
```

```
            | sign term
            | simple_expression adding_operator term
            ;

simple_name : identifier
            | TNULL  /* NOTE: this is a quick fix to simplify the
  waveform_element rule. */
            ;

slice_discrete_range  /* currently restricted to integer subrange */
            : unsigned_integer direction unsigned_integer
    ;

slice_name  : prefix TLPAREN slice_discrete_range TRPAREN
            ;

stateattribute
            : is_chip     /* ...in any order */
            | binding
            | is_module
            ;

stateattributes
            : stateattribute   /* any number of attributes can exist... */
            | stateattributes stateattribute
            ;

stateattributes_opt           /* all attributes are optional */
            : stateattributes
            | empty
            ;

statement   :
              assert_statement
            | case_statement
            | comment_statement
            | else_statement
            | elsif_statement
            | endcase_statement
            | endif_statement
            | endloop_statement
            | exit_statement
            | forloop_statement
            | if_statement
            | loop_statement
            | null_statement
            | procedure_call_statement
            | return_statement
            | signal_assignment_statement
            | variable_assignment_statement
            | wait_statement
            | when_statement
            | whileloop_statement
            ;

statements
            : statement
```

The following describes the tokens that were used throughout the above syntax description.

```
A       [aA]
B       [bB]
C       [cC]
D       [dD]
E       [eE]
F       [fF]
G       [gG]
H       [hH]
I       [iI]
J       [jJ]
K       [kK]
L       [lL]
M       [mM]
N       [nN]
O       [oO]
P       [pP]
Q       [qQ]
R       [rR]
S       [sS]
T       [tT]
U       [uU]
V       [vV]
W       [wW]
X       [xX]
Y       [yY]
Z       [zZ]
```

```
letter                  [a-zA-Z]
digit                   [0-9]
identifier              {letter}("_"?({letter}|{digit}))*
unsigned_integer        ({digit})+
exponent                E[+-]?{unsigned_integer}
extended_digit          {digit}|[a-fA-F]
based_integer           ({extended_digit})+("_"?({extended_digit})+)*
```

```
{B}{I}{T}                       { return(TBIT); }
{B}{I}{T}"_"{V}{E}{C}{T}{O}{R}  { return(TBITVECTOR); }
{B}{O}{O}{L}{E}{A}{N}           { return(TBOOLEAN); }
{C}{H}{A}{R}{A}{C}{T}{E}{R}     { return(TCHARACTER); }
{I}{N}{T}{E}{G}{E}{R}           { return(TINTEGER); }
{N}{A}{T}{U}{R}{A}{L}           { return(TNATURAL); }
{P}{O}{S}{I}{T}{I}{V}{E}        { return(TPOSITIVE); }
{R}{E}{A}{L}                    { return(TREAL); }
{S}{T}{R}{I}{N}{G}              { return(TSTRING); }
{T}{I}{M}{E}                    { return(TTIME); }
```

```
{S}{T}{A}{T}{E}  return(TSTATE);
{N}{A}{M}{E}  return(TNAME);
{D}{E}{C}{L}{A}{R}{A}{T}{I}{O}{N}{S}              return(TDECLARATIONS);
{C}{O}{N}{N}{E}{C}{T}{I}{O}{N}{S}  return(TCONNECTIONS);
{E}{S}{T}{I}{M}{A}{T}{E}{S}   return(TESTIMATES);
{C}{O}{N}{S}{T}{R}{A}{I}{N}{T}{S}   return(TCONSTRAINTS);
{I}{S}"_"{C}{H}{I}{P}                              return(TIS_CHIP);
{I}{S}"_"{M}{O}{D}{U}{L}{E}                        return(TIS_MODULE);
{C}{O}{D}{E}                                       return(TCODE);
{S}{E}{Q}{U}{E}{N}{T}{I}{A}{L}[ \t]*{S}{U}{B}{S}{T}{A}{T}{E}{S}
return(TSEQUENTIAL_SUBSTATES);
{C}{O}{N}{C}{U}{R}{R}{E}{N}{T}[ \t]*{S}{U}{B}{S}{T}{A}{T}{E}{S}
return(TCONCURRENT_SUBSTATES);

{A}{S}{S}{E}{R}{T}   return(TASSERT);
{R}{E}{P}{O}{R}{T}   return(TREPORT);
{S}{E}{V}{E}{R}{I}{T}{Y}   return(TSEVERITY);
"--"          return(TCOMMENT);
"=>" return(TARROW);
"\|" return(TBAR);
{O}{T}{H}{E}{R}{S} return(TOTHERS);

{C}{A}{S}{E} return(TCASE);
{C}{H}{A}{N}{N}{E}{L} return(TCHANNEL);
{C}{O}{N}{S}{T}{A}{N}{T} return(TCONSTANT);
{D}{E}{F}{I}{N}{E} return(TDEFINE);
{F}{U}{N}{C}{T}{I}{O}{N}                          return(TFUNCTION);
{P}{O}{R}{T} return(TPORT);
{P}{R}{O}{C}{E}{D}{U}{R}{E} return(TPROCEDURE);
{S}{I}{G}{N}{A}{L} return(TSIGNAL);
{S}{U}{B}{T}{Y}{P}{E} return(TSUBTYPE);
{T}{Y}{P}{E} return(TTYPE);
{V}{A}{R}{I}{A}{B}{L}{E} return(TVARIABLE);
{I}{S} return(TIS);
{B}{E}{G}{I}{N} return(TBEGIN);
{R}{E}{T}{U}{R}{N} return(TRETURN);
{I}{N} return(TIN);
{O}{U}{T} return(TOUT);
{I}{N}{O}{U}{T} return(TINOUT);

{B}{U}{S}                                         return(TBUS);
{R}{E}{G}{I}{S}{T}{E}{R}                          return(TREGISTER);

{W}{A}{I}{T} return(TWAIT);
{I}{F}          return(TIF);
{T}{H}{E}{N} return(TTHEN);
{E}{L}{S}{I}{F} return(TELSIF);
{E}{L}{S}{E} return(TELSE);
{L}{O}{O}{P} return(TLOOP);
{E}{N}{D}                                         return(TEND);
{N}{U}{L}{L}       return(TNULL);
{A}{F}{T}{E}{R} return(TAFTER);
{T}{R}{A}{N}{S}{P}{O}{R}{T} return(TTRANSPORT);
{O}{N} return(TON);
{U}{N}{T}{I}{L} return(TUNTIL);
{F}{O}{R} return(TFOR);
```

```
 {W}{H}{I}{L}{E} return(TWHILE);
 {E}{X}{I}{T} return(TEXIT);
 {W}{H}{E}{N} return(TWHEN);
 {R}{A}{N}{G}{E} return(TRANGE);
 {A}{R}{R}{A}{Y} return(TARRAY);
 "<>" return(TBOX);
 {O}{F} return(TOF);

 {N}{O}{T} return(TNOT);
 {A}{B}{S} return(TABS);
 {M}{O}{D}                                    return(TMOD);
 {R}{E}{M}                                    return(TREM);

 {A}{N}{D} return(TAND);
 {O}{R}    return(TOR);
 {N}{A}{N}{D} return(TNAND);
 {N}{O}{R}    return(TNOR);
 {X}{O}{R}    return(TXOR);

 {T}{O} return(TTO);
 {D}{O}{W}{N}{T}{O} return(TDOWNTO);

 {S}{T}{A}{B}{L}{E} return(TSTABLE);
 {E}{V}{E}{N}{T} return(TEVENT);
 {Q}{U}{I}{E}{T} return(TQUIET);
 {A}{C}{T}{I}{V}{E} return(TACTIVE);
 {D}{E}{L}{A}{Y}{E}{D} return(TDELAYED);
 {L}{E}{N}{G}{T}{H} return(TLENGTH);
 {L}{O}{W} return(TLOW);
 {H}{I}{G}{H} return(THIGH);

 {T}{I}{M}{E}{O}{U}{T} return(TTIMEOUT);
 {O}{T}{H}{E}{R} return(TOTHER);

 {identifier}"\_"{F}{C}{T}                    return(TFCTNAME);
 {identifier}"\_"{P}{R}{O}{C}                 return(TPROCNAME);

 {E}{X}{P}{R}{F}{O}{R}{P}{A}{R}{S}{E} return(TEXPRFORPARSE);
 {S}{T}{M}{T}{F}{O}{R}{P}{A}{R}{S}{E} return(TSTMTFORPARSE);
 {D}{E}{C}{L}{F}{O}{R}{P}{A}{R}{S}{E} return(TDECLFORPARSE);

 {identifier}                                 return(TIDENTIFIER);
 {unsigned_integer} return(TUINT);
 ({unsigned_integer}"\."{unsigned_integer}{exponent}) |
 ({unsigned_integer}"\."{unsigned_integer}) |
 ({unsigned_integer}{exponent})   return(TFLOAT);

 ({unsigned_integer}"\#"{based_integer}"\."{based_integer}"\#"{exponent}) |
 ({unsigned_integer}"\#"{based_integer}"\."{based_integer}"\#") |
 ({unsigned_integer}"\#"{based_integer}"\#"{exponent})
 return(TBASEDLIT);

 ("\'"(({letter})|({digit}))"\'")  return(TCHARLIT);
 ("\""(.)*"\"")   return(TSTRINGLIT);
 ("B""\""{based_integer}"\"") |
 ("O""\""{based_integer}"\"") |
 ("X""\""{based_integer}"\"")     return(TBSTRINGLIT);
```

```
";"                                     return(TSEMICOLON);
":"                                     return(TCOLON);
","                                     return(TCOMMA);
"."                                     return(TPERIOD);
"{"                                     return(TLCURL);
"}"                                     return(TRCURL);
"["                                     return(TLBRACK);
"]"                                     return(TRBRACK);
"("                                     return(TLPAREN);
")"                                     return(TRPAREN);
"'" return(TAPOSTROPHE);


"+"                                     return(TPLUS);
"-"                                     return(TMINUS);
"&" return(TCONCAT);
"*"                                     return(TTIMES);
"/"                                     return(TDIVIDE);
"**"                                    return(TPOWER);

"="                                     return(TEQUAL);
"/="                                    return(TNEQUAL);
"<"                                     return(TLT);
"<="                                    return(TLTE);
">"                                     return(TGT);
">="                                    return(TGTE);


":="                                    return(TVARASSIGN);


 /* C comment */
"/*" { register int c;
                                    while ((c = input())) {
                                       if (c == '*') {
                                          if ((c = input()) == '/')
                                             break;
                                          else
                                             unput (c);
                                       }
                                       else if (c == '\n')
                                          Line_no++;
                                       else if (c == 0)
                                          commenteof();
                                    }
}

[ \t\f]                             ;

\n                                  Line_no++;

.                                   { fprintf (stderr,
                                    "'%c' (0%o): illegal character at line %d\n",
                                    yytext[0], yytext[0], Line_no);
                                    }
```